SASL プラグイン開発メモ

01 版　2024/04/30　　初版　　By Tanzai

目次

はじめに

この資料は、X-Plane をカスタマタイズする為の SASL を使ったプラグインの作成方法について解説します。なお SASL の記述言語である Lua についての解説は省かれています。この資料は、SASL 用記述の解説及び実際の X-Plane への応用例も記載します。ただしそれら応用例は動作保証するものではありません、読者本人の責任の元に参照してください。
この様な資料は X-Plane ソフト開発グループで共有し、ほとんど一般には公開されていない様です。この機会に SASL を使った X-Plane 用機体の高機能化に興味を持っていただければ幸いです。内容に誤りがあればご指摘よろしくお願いします。

なお、この資料は以下のアプリを使用しています。
・SASL v3.16.4, 64 bit only
・X-Plane 11.4 (64 bit)
・Plane Maker 11.4　英語表示
・Blender 2.8
・XPlane2Blender 3.5
・Gimp 2.10
これらのアプリの使い方詳細については、この資料では述べません。

1. プラグイン概要

1.1 Lua 言語プラグインの概要

Lua 言語を使った X-Plane 用プラグインには以下の様な複数の方式が有ります。

FlyWithLua
Xlua
SASL

これらは Lua 言語の細部の記述が微妙に異なっており、それぞれ専用のマニュアルを参照し開発する必要があります。
これら方式は動作原理は良く似ています。X-Plane の動作開始直後にプラグインは決められた Lua ファイルを読み取り、X-Plane 実行中のどの条件の時に何を実行するか取得します。X-Plane 起動後にその条件が発生したかを監視し、その条件が検出されると決められた動作を実行します。

FlyWithLua で作られたプラグインは、X-Plane の Resources フォルダにインストールされ、X-Plane 全体的なカスタマイズに使われます。個々の機体のカスタマイズには不向きです。

Xlua で作られたプラグインは、個々の機体フォルダ内にインストールされ、その機体の特性をカスタマイズします。基本的に既存の Dataref の値を Lua スクリプトで操作するものです。詳しいマニュアルが無いので、複雑な修正を行うプラグインは開発が困難な様です。

SASL で作られたプラグインは、Dataref 値の操作のみならず X-Plane 画面の操作や入力方法など多様なカスタマイズをすることが出来ます。この資料では機体フォルダに保存する SASL プラグインについて解説します。

これらはフリーソフトで開発したプラグインは基本的に自由に配布出来ます。

1.2 SASL プラグインについて

SASL とは Scriptable Avionics Simulation Library の略で詳しい使い方は
https://1-sim.com/ のページから **SASL v3 manual** を参照します。

この方法で作られたプラグインは X-Plane 内のいくつかの場所にインストールされます。

X-Plane 11/Resources/plugins/ フォルダ内にインストールされる場合（Global プロジェクト）は X-Plane の全般的な動作をカスタマイズします。

シーナリフォルダ内にインストールされる場合 (Scenery プロジェクト) はシーナリ動作をカスタマイズします。

X-Plane 11/aircraft/の機体フォルダ内の plugins フォルダにインストールされる場合（Aircraft プロジェクト）は各機体の動作をカスタマイズします。この資料ではこの場合について解説します。

SASL プラグインは独立して動作します。処理が互いに干渉しないのであれば複数のプラグインを同時に実行することも出来ます。（SASL と Xlua の混在も可能）
また SASL プラグインは文字や図形を描いたり、音を再生するなど色々カスタマイズが可能です。詳細は 3 項を参照して下さい。

無料で配布する 場合は無料 SASL を使用します、これには有料 SASL の様に暗号化やシリアルキーは使えません。

1.3 SASL プラグインのファイル構成

X-Plane を起動するとプラグイン内の main.lua と呼ばれるファイルが最初に読み取られます。このファイルには X-Plane 画面の表現内容や X-Plane 実行中に決められた場面(event)で実行したい動作(callback) などが定義されます。
また一部の作業はその機能を別 Lua ファイル(component) に記載し運用することも可能です。この場合は main.lua 文にその旨記載します。
その場合は変数の扱いに必要が注意です。何も指定がなければ原則として変数は記述されている Lua ファイル内でのみ有効です。main.lua や配下の component の間で使用する場合、または別プラグインや Plane Maker でも使用する場合は、2.4 項に従って変数を定義します。その場合変数の参照には get 関数、代入には set 関数が必要となります。

2. プラグイン作成

2.1 準備

SASL プラグインの原型は https://1-sim.com/ のページから **SASLFree** フォルダをダウンロードします。

```
/ SASLFree /
    ├ / 64 /
    ├ / data /
    ├ / liblinux /
    ├ changelog.txt
    ├ functions.txt
    └ version.txt
```

対象の機体フォルダに plugins フォルダを作成しその中に上記 SASLFree フォルダを保存します。その後 SASL プラグイン開発で基本的に以下の部分の修正又はファイル追加を行う必要が有ります。

(1) SASLFree を作成するプラグインを示す名称に変更
この名称は他の人が作成するプラグインと重複してはいけません。

(2) configuration 内の configurationExample.ini ファイルを修正
Text エディタでこのファイルを修正し、最後に configuration.ini として保存します。このプロジェクトが Global であるか Scenary であるか Aircraft であるか等の指定します。(2.2 項参照)

(3) オブジェクトや画像やフォント等のファイルを保存
configuration 内にこのプロジェクトで使用するオブジェクトや画像やフォント等を保存します。ここにファイルを追加した場合は、そのファイルを呼び出すときにサーチパス追加の宣言が必要。(3.2 項の例)

(4) modules 内に main.lua を保存
このファイルはプラグイン固有の変数や関数の定義、入力方法指定、及び component（別 lua ファイル）の指定を行います。(2.3 項参照)
X-Plane 起動直後、シーナリーや機体データが読み取られる前に、この main.lua が読み込まれ内容が SASL で解析されます。
X-Plane 動作中は定義されたタイミング (event) が監視され条件がそろえば関数(callback) が実行され X-Plane 動作が修正されます。

(5) Custom Module 内に別 Lua ファイル (component) を追加
メインのデータ処理に定型処理がある場合、別の Lua スクリプト (component と呼ばれる) を作成する場合があります。そのファイルは Custom Module フォルダに保存します。

最終的に以下の様な構成になります。

Aircraft プロジェクトのファイル構成例：

```
/Any_Aircraft/                          <-- プラグインを追加する機体フォルダ
  :
  ├ / plugins /                         <-- このフォルダを新設する
  │   └ / SASLFree /                    <-- この名称はプラグインの名称に変更する
  │       ├ / 64 /
  │       ├ / data /
  │       │   ├ / api /
  │       │   ├ / components /
  │       │   ├ / init /
  │       │   ├ / modules /
  │       │   │   ├ / 3rd-modules /
  │       │   │   ├ / configuration /
  │       │   │   │   ├ widgetResources
  │       │   │   │   ├ configuration.ini      <-- 修正必要
  │       │   │   │   └ 画像ファイルなど       <-- 必要に応じ追加する
  │       │   │   ├ / Custom Module /
  │       │   │   │   ├ (任意名称).lua        <-- 必要に応じ追加する
  │       │   │   :
  │       │   │   └ main.lua                  <-- 必ず作成要
  │       │   └ / output /
```

2.2 configuration.ini 作成

ダウンロードした configurationExample.ini の [project] 部分をテキストエディタで必要に応じて以下の内容を修正し、configuration.ini の名前で保存します。

Id: この項目は使用せず(任意の数字を指定)

name: このプラグインの名称を指定します。この名称はログインデータで参照されます。

type: プロジェクトタイプ番号を指定します
- 0: Aircraft project
- 1: Scenery project
- 2: Global project

startDisabled: X-Plane 起動時にこのプラグインを使うので通常 0 を指定します。

widget: Widget をつかうか否かを指定します。
- 0: Widget は使わない（正式リリース時に設定する）
- 1: Widget を使う（デバック中に設定する）

Aircraft プロジェクトの configuration.ini 例

```
[project]
   id=1
   name=Person1
   type=0
   startDisabled=0
   widget=0

[sceneryProject]
   centerLatitude=0.0000000
   centerLongitude=0.0000000
   maxElevation=0.0
   radius=0.0

[widget]
   font=SourceCodePro.ttf
        （以下省略）
```

} Aircraft プロジェクトの場合不要

2.3 main.lua 作成

X-Plane を起動すると機体に搭載されているプラグインの main.lua ファイルが、シーナリや機体データよりも先に読み取られます。main.lua の名称は変更出来ません。
記載の順番は最初にプラグインで使われる画面、変数やその初期値等を定義し、その後処理内容を記述した function 文を記述します。定義する前にそれら変数や関数等を main.lua (または component) に記載するとエラーになります。
main.lua は概ね以下の記述から構成されます。

(1) 必要な記述
冒頭に下記 3 行を記入します。入れないと動作が変になる場合が有りました。

```
setAircraftPanelRendering(false)
set3DRendering(false)
setInteractivity(false)
```

注 プラグインの入力操作がおかしい場合はここを true にしてみて下さい。

(2) プラグインの component 構成を定義
SASL プラグインは main.lua の他に別.Lua ファイル(component) に分割して構成することが出来ます。(1.3 項)
例えば main.lua で以下の文を記載した場合は、

```
components = { textArea1{ }, textArea2{ } }
```

textArea1.lua および textArea2.lua というファイルが main.lua 配下に存在する事を定義しています。拡張子抜きのファイル名が component 名称となります。なお .lua ファイルは Custom Module フォルダに格納する必要あります。

(3) 画面上の入出力するエリアを定義
画面に文字等を表現する場合やマウス操作する場合は、そのエリアを contextWindow で定義し、component (main.lua 以外の lua ファイル) にその詳細を記述します。その component は Custom Module フォルダに格納します。
例えば infoBoard.lua という component で X-Plane 画面の数の領域について定義する場合、main.lua に以下のように記載します。(2.6 項参照)

(3) 変数定義
英文では Property と記載されていますがこの資料では変数と訳します。プラグイン内の Lua ファイルが扱う変数は、そのままではその Lua ファイル内でしか使えません。プラグインの他の Lua ファイルでも使う場合、または X-Plane の専用変数(dataref) を流用する場合等は、その旨 main.lua の冒頭で宣言する必要あります。詳しくは 2.4 項を参照下さい。

(4) function 設定
纏まった処理を function 文に記載します。X-Plane 実行中に条件(event) が揃えば実行される function は callback と呼ばれています。詳細は 2.5 項参照。

(5) 入力方法の定義
X-Plane 実行中にユーザが行う操作手段が幾つかあります。その操作と実行内容を定義します。

① 画面上部のメニューによる入力
X-Plane 画面上部の Plugins にメニューを追加できます。 詳しくは 3.4 項を参照下さい。

② キー入力設定
キーボードのキー入力で起動する関数を定義できます。詳しくは 3.6 項を参照下さい。

③ マウス入力設定
詳しくは 3.5 項参照

2.4 変数について

SASL で使用出来る変数名は Lua 言語の変数名条件と同じです。すなわち、英数字で、大文字小文字を区別し先頭文字は数字以外で、記号は _（アンダーライン）のみ使用可能となります。更に Lua や SASL の予約語は使えません。
その他以下の様な SASL 特有の記載が必要となり、定義，引用、代入に注意が必要となります。

(1) Local 変数
変数名の前に local を付記し宣言します。その宣言文の次の文から do, if 文などのブロックの最後まで有効な変数となります。値や他の変数の代入には **=** を使います。

定義、代入 ： local〈変数名〉**=**〈変数値または変数名〉

(2) Global 変数
特に宣言しないで使う変数はその Lua ファイル内でのみ有効です。値などの代入には **=** を使います。

定義 ： 定義は不要です。
代入 ： 〈変数名〉**=**〈変数値または変数名〉

(3) 他の component（他の Lua ファイル) でも有効な変数
defineProperty で定義します。他の Lua ファイルから引用できる様になります。引用には get 文、代入には set 文を使用します。

定義： defineProperty ("〈変数名〉**" ,** 〈初期値〉)
例）defineProperty ("mode_ID" , 0)

引用： 〈代入する変数名〉= get (〈変数名〉)
例）current_mode = get (mode_ID)

代入： set ("〈変数名〉**" ,** 〈変数値〉)
例）set (mode_ID , 1)

(4) dataref の値を引用する変数

X-Plane 内部変数である dataref は globalProperty で変数を定義します。
その変数の型や変数か配列であるかで扱い方法が少し異なります。

dataref が一つの変数である場合の定義：

単精度整数　：〈変数名〉= globalPropertyi (" sim/.../… ")
単精度実数　：〈変数名〉= globalPropertyf (" sim/.../… ")
倍精度実数　：〈変数名〉= globalPropertyd (" sim/.../… ")
文字列　　：〈変数名〉= globalPropertys (" sim/.../… ")
（以下省略）

例）winW = globalPropertyi("sim/graphics/view/window_width")

引用：〈代入する変数名〉= get(〈変数名〉)
代入：set (〈変数名〉, 〈代入する値〉)

dataref が配列である場合の定義：

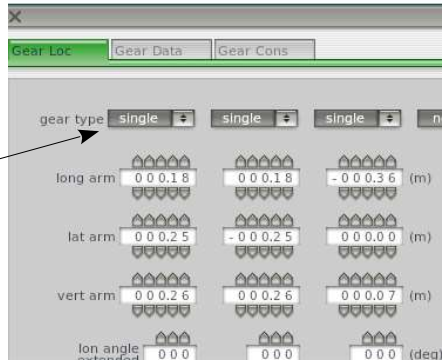
整数配列　　：〈変数名〉= globalPropertyia (" sim/.../… ")
実数配列　　：〈変数名〉= globalPropertyfa (" sim/.../… ")
例）tire_ang = globalPropertyfa
("sim/flightmodel2/gear/tire_steer_actual_deg")

引用：〈代入する変数名〉= get (〈変数名〉, 〈Id〉)
例）current_ang = get (tire_ang , 3)
tire_ang で定義した dataref の 3 番目の要素を current_ang に代入

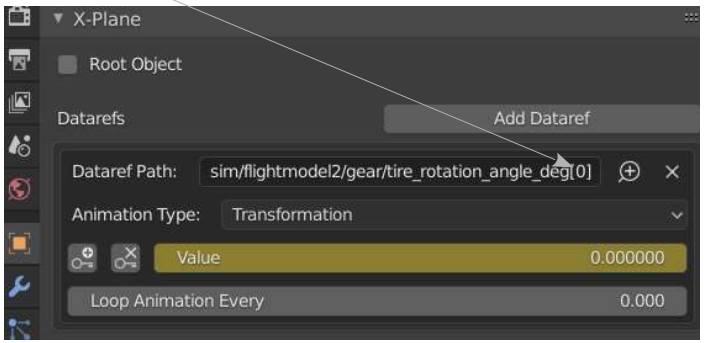
代入：set (〈変数名〉, 〈代入する値〉, 〈Id〉)
例）set (tire_ang , 15 , 3)
tire_ang として定義した dataref の 3 番目の要素に 15 を代入

ここにはクリエータを混乱させる Plane Maker と SASL 間の番号ズレ問題があります。

例えば Plane Maker で車輪を定義した場合、その最初の (左端の) 車輪は 0 番となります。



そのため最初の車輪は Blender のアニメーションでは下図の様に[0]を指定します。



一方、SASL ではこの車輪を 1 番として扱います。例えば Main.lua で 1 番の車輪 (Blender では 0 番の車輪) の回転で音を出す場合は、以下の様に記述します。

```
local tire_ang =
globalPropertyfa("sim/flightmodel2/gear/tire_rotation_angle_deg")
sound_shoes0 = sasl.al.loadSample("sound_shoes0.wav")
：
function update()
   current_ang = get(tire_ang, 1)
   n = 0
   if current_ang < 10 then
      n = 0
   elseif (n == 0) and (current_ang > 105) then
      sasl.al.playSample(sound_shoes0)
      n = 1
   end
end
```

ソフト屋さんの認識不足からこうなったと思いますが、今となってはこのまま使うしかありません。以下の様に覚えましょう。

PlaneMaker / Blender は 0 から、SASL は 1 から

dataref が配列である時の特定要素だけを定義：
整数配列要素：〈変数名〉= globalPropertyiae (" sim/.../..." , 〈Id〉)
実数配列要素：〈変数名〉= globalPropertyfae (" sim/.../..." , 〈Id〉)
（以下省略）

引用：〈代入する変数名〉= get (〈変数名〉)
代入：set (〈変数名〉, 〈代入する値〉)

(5) dataref を新設する場合

createGlobalProperty.. で変数を定義します。他 Lua ファイル(component)のみならず他のプラグインからでも引用できます。変数名は / で区切られた名称を使います。ただし sim/... の名称は X-Plane が利用するので使えません。また他のプラグインが定義する名称と重なってはいけません。変数の値が整数か実数かまたは数列かで扱いが一部異なります。

dataref が一つの変数である場合の定義：
単精度整数　：〈変数名〉= createGlobalPropertyi (**"** 新設 dref ")
単精度実数　：〈変数名〉= createGlobalPropertyf (" 新設 dref ")
倍精度実数　：〈変数名〉= createGlobalPropertyd (" 新設 dref ")
文字列　　：〈変数名〉= createGlobalPropertys (" 新設 dref ")

例）check_dref1 = createGlobalPropertyi ("Woman1/livery/Id" , 0)
Plane Maker で killdataref を使う場合にこの様な定義をします。
(3.12 項参照)

引用：〈代入する変数名〉= get (〈変数名〉)
代入：set (〈変数名〉**,** 〈代入する値〉)
例）set (check_dref1 , 0)

dataref が配列である場合の定義：
整数配列　　：〈変数名〉= createGlobalPropertyia (" 新設 dref ")
実数配列　　：〈変数名〉= createGlobalPropertyfa (" 新設 dref ")

引用：〈代入する変数名〉= get (〈変数名〉**,** 〈Id 配列〉)
代入：set (" 新設 dref **",** <代入する値又は変数> **,** 〈Id 配列〉)

（以下省略）

2.5 関数について

function 文で制御内容のまとまりを宣言します。関数が引用される前にはその関数が宣言されていなければなりません。

定義：　function ＜名前＞(引数リスト)
　　：
　　end

引用方法 1：　変数 = function ＜名前＞(引数リスト)
引用方法 2：　function ＜名前＞(引数リスト)

＜名前＞の命名条件は変数の場合と変わりません。

ただし下表の function 名称は callback 条件が決められています。

関数名称	callback の条件
onSceneryLoaded ( )	シーナリーと機体を指定し飛行開始すると最初にこの関数が実施されます。その後シーナリーや機体を変更する度に実施されます。
onAirportLoaded ( )	シーナリーと機体を指定し飛行開始すると最初にこの関数が実施されます。その後シーナリーや機体を変更する度に実施されます。 機体が空港の所定の場所に移動されます。
onModuleInit ( )	シーナリーと機体を指定し飛行開始すると最初だけこの関数が実施されます。
update ( )	X-Pane 実行中 1 フレーム描画毎に関数内の変数を監視し必要あれば処理を実行します。 この関数実施と同時に配下の Component でも変更が必要な場合は、UpdateAll を実施する必要があり書き込みます。
draw ( )	前記 onModuleInit( ) から最初の update( ) を実施するまでの間のみ 2D 画像を書き込みます。 SASL の処理速度を確保するために関数内では演算処理をしないで下さい。if 文等は可能です。この関数実施と同時に配下の Component でも書き込みが必要な場合、DrawAll を実施する必要があります。
draw3D ( )	前記 onModuleInit( ) から最初の update( ) を実施するまでの間のみ 3D 画像を書き込みます。X-Plane オブジェクトは書き込めません。 SASL の処理速度を確保するために関数内では演算処理をしないで下さい。if 文等は可能です。この関数実施と同時に配下の Component でも書き込みが必要な場合、DrawAll3D を実施する必要があります。
drawObjects ( )	前記 onModuleInit( ) から最初の update( ) を実施するまでの間のみ X-Planee オブジェクトを書き込みます。 SASL の処理速度を確保するために関数内では演算処理をしないで下さい。if 文等は可能です。この関数実施と同時に配下の Component でも書き込みが必要な場合、DrawAllObjects を実施する必要があります。
onPlaneLoaded ( )	機体がロードされた時にこの関数が実行される様です。
onPlaneUnloaded ( )	SASL を使用する機体が変更されアンロードされた時にこの関数を実施されます。
onModuleShutdown ( )	SASL を使用する機体が変更された時に onPlaneUnloaded( ) と onModuleDone( )の間に実行されます。（*1）
onModuleDone ( )	SASL を使用する機体が変更された時に最後にこの関数が実行されます。（*1）
onAirplaneCountChanged ( )	機体の数を変更した時にこの関数が実行される様です。
onPlaneCrash ( )	クラッシュ時はディフォルトで SASL の再起動が行われますが、この関数で 0 をリターンするとクラッシュ時に再起動処理は行われなくなります。

*1 機体を終了した場合は実行されますが。X-Plane を終了した時は実行されません。

2.6 contextWindow( ) について

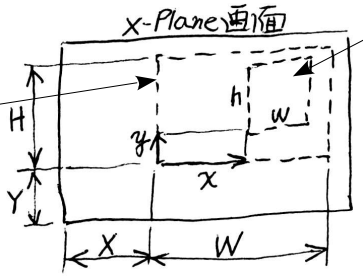
contextWindow は X-Plane 画面上に文字、画像、図形等を表示したり、マウス操作エリアを設定したりするので良く使われます。
以下に contextWindow の各パラメータとそのディフォルト値を示します。ディフォルト以外の設定が必要な場合のみそのパラメータを記載します。ただし position だけは記載必須です。

```
＜呼び名＞ = contextWindow {
    name            = " ＜画面名＞ " ,                          (*1)
    position        = {     X , Y , W , H } ,                   (*2)
    minimumSize     = { 100 , 100 } ,                           (*3)
    maximumSize     = { 2048 , 2048 } ,                         (*3)
    visible         = false ,                                   (*4)
    proportional    = true ,                                    (*3)
    gravity         = { 0 , 1 , 0 , 1 } ,                       (*5)
    noBackground    = false ,                                   (*6)
    layer           = SASL_CW_LAYER_FLOATING_WINDOWS ,          (*7)
    noDecore        = false ,                                   (*1)
    customDecore    = false ,                                   (*1)
    noResize        = false ,                                   (*3)
    resizeMode      = SASL_CW_RESIZE_ALL_BOUNDS ,               (*8)
    noMove          = false ,                                   (*9)
    saveState       = false ,                                   (*10)
                :
          （その他省略）
                :
    components = { ＜component 名＞ { position = { x , y , w , h }, } , } ,   (*11)
}
```

*1 noDecore = false および customDecore = false を指定すると標準の画面が表示されますが、その時に表示されるタイトル名を name で指定します。

*2 position は指定必須のパラメーターです。
X-Plane 画面の左下を原点とし以下を設定します。

X　：この contextWindow の左端の x 座標
Y　：この contextWindow の下端の y 座標
W　：この contextWindow の幅
H　：この contextWindow の高さ



*3 noResize = true とするとサイズ変更可能となり、その最小および最大は minimumSize および maximumSize で制限される。

*4 contextWindow は visible = true で表示される。

*5 gravity = { 0 , 1 , 0 , 1 } は下方が重力方向である事を示す。

*6 noBackground = true とするとディフォルトの背景色になります。

*7 画面のレイーヤ位置を指定します。

SASL_CW_LAYER_FLIGHT_OVERLAY	低
SASL_CW_LAYER_FLOATING_WINDOWS	▲
SASL_CW_LAYER_MODAL	▼
SASL_CW_LAYER_GROWL_NOTIFICATIONS	高

*8 resizeMode リサイズ方式の設定します。

SASL_CW RESIZE_ALL_BOUNDS	: 全エッジ移動可
SASL_CW_RESIZE_RIGHT_BOTTOM	: 右下隅のみ移動可

*9 noMove = true で contextWindow は位置固定されます。

*10 saveState = true でプロジェクト終了時に contextWindow の状態を保存します。

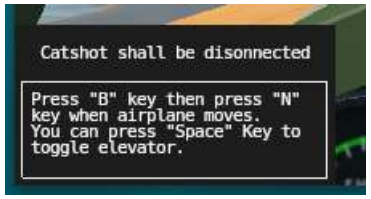
*11 components はこの contextWindow で動作する component を列挙します。
列挙された comonent には指定必須のパラメーターである position を指定します。

x　：追加された component の左端の x 座標
y　：追加された component の左端の y 座標
w　：追加された component のエリアの幅
h　：追加された component のエリアの高さ

3. 作成例

3.1 messageWindow を追加

messageWindow による表示パネルは X-Plane 画面に右図の様な表示となります。



指定方法：
(1) ボタンを入れない場合
sasl.messageWindow (x, y, w, h, title , message, 0, lifetime)

(2) ボタン(N 個) を入れる場合
sasl.messageWindow (x, y, w, h, title , message, N, buttonName1, callback1, buttonName2, callback2, … , buttonNameN, callbackN)

引数の意味

x	: messageWindow 開始点の x 座標
y	: messageWindow 開始点の x 座標
w	: messageWindow の幅
h	: messageWindow の高さ
title	: messageWindow のタイトル
Message	: 画面に表示する文またはその呼び名
N	: 付加するボタンの数
lifetime	: 画面の表示時間(sec)
buttonName1	: 最初のボタン名
callback1	: 最初のボタンを押した時に実行する callback
buttonName2	: 二番目のボタン名
callback2	: 二番目のボタンを押した時に実行する callback

右上画面の指定例:
最初に main.lua で X-Plane 画面左下の隅から幅 400, 高さ 100 の contextWindow を定義し、その領域内に infoBoard.lua で指定するエリアを定義しています。同時に表示切替のタイミングを与える変数 postCarr と catStatus を宣言します。
別 lua ファイル(infoBoard.lua) にて所定の条件で文字を表示します。

main.lua への記載

```
local popup = contextWindow {
   name                = "popup",
   position            = {0, 0, 400, 100},
   noResize            = true,
   visible             = true,
   noBackground        = true,
   layer          = SASL_CW_LAYER_GROWL_NOTIFICATIONS,
   noDecore            = true,
   noMove              = true,
   components  = { infoBoard { position = { 0, 0, 400, 100},},},
}

defineProperty ( "postCarr" , 0 )
defineProperty ( "catStatus" , 0 )
```

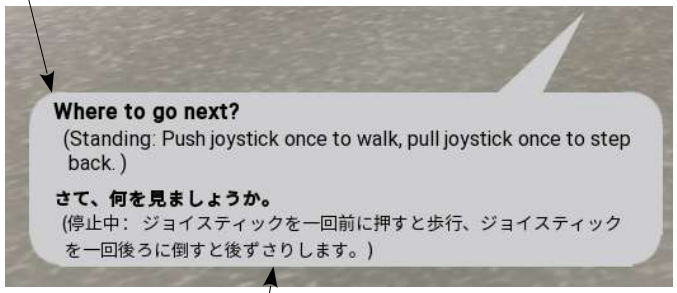
次に子 Lua ファイル(この例では infoBoard.lua) の冒頭で表示する文字列を定義しその後 sasl.messageWindow ( ) 文で文字を表示します。

infoBoard.lua での記述

```
title1 = " Catshot shall be disonnected "
title2 = " Catshot is disconnected "
title3 = " ZUIKAKU is not ready "
text1 = "Press \"B\" key then press \"N\" key when airplane moves.    You can press \"Space\" Key to toggle elevator."
text2 = "Now you can take off.                                    Pressing \"N\" Key resets aircraft position."
text3 = "Select ZUIKAKU in     \"plugins / Set carrier to\" menu."

function draw()
   if postCarr == 1 then
      if catStatus == 1 then
         sasl.messageWindow ( 0, 0, 200, 100, title1, text1, 0, 5 )
      elseif catStatus == 0 then
          sasl.messageWindow ( 0, 0, 200, 100, title2, text2, 0, 5 )
      end
   else
      sasl.messageWindow ( 0, 0, 200, 100, title3, text3, 0, 5 )
   end
end
```

3.2 2D Graphics 機能による表示

X-Plane 画面に drawTexture ( ) で画像を貼り付け、drawText ( ) で文字を表示できます。最初に main.lua で contextWindow によりテクスト画面を宣言し、次に子 Lua ファイルで貼りつける画像や文字を指定します。画像ファイル(例えば下図の吹き出し) はプラグインの /configuration/ フォルダに格納しておきます。



以下の例では main.lua で、X-Plane 画面左下を基準とし x=20, y=20 の点から幅 500 高さ 200 のエリアに popup2 と云う contextWindow を定義し、その領域内の x=0, y=0 の点から幅 500dot 高さ 200dot のエリアを使って textArea2.lua により画像や文字を指定しています。

main.lua への記載例

```
-- main.lua
local popup2 = contextWindow {
        name                = "popup2",
        position            = {20, 20, 500, 200},
        noResize            = true,
        visible             = true,
        noBackground        = true,
        noDecore            = true,
        noMove               = true,
        components          = {textArea2{position={0, 0, 500, 200},},},
}
```

次に textArea2.lua を作成し custom Module/フォルダに格納します。
その中で記載文字列 (E1, E2, J1, J2) を定義し、draw ( ) 関数の中で drawText を使ってそれら文字列を表示します。

textArea2.lua への記載例

```
-- textArea2.lua
addSearchResourcesPath( moduleDirectory .. "/configuration/" )
local tex1 = sasl.gl.loadImage ( "popup.png" , 0 , 0 , 500 , 200 )
local roboto = loadFont (getXPlanePath( ) .. "Resources/fonts/Roboto-Regular.ttf")
local noto = loadFont (getXPlanePath( ) .. "Resources/fonts/NotoSansCJK-SC-Regular.otf")
local black = {0, 0, 0, 1}
E1 = "Where to go next?"
E2 = "  (Standing: Push joystick once to walk, pull joystick once to step\n   back. )"
J1 = "さて、何を見ましょうか。"
J2 = "  (停止中 ： ジョイスティックを一回前に押すと歩行、ジョイスティック\n   を一回後ろに倒すと後ずさりします。)")

function draw ( )
   sasl.gl.drawTexture( tex1 , 0 , 0 , 500 , 200)
   drawText(roboto ,20 ,116 , E1 ,16 ,true ,false ,TEXT_ALIGN_LEFT ,black )
   drawText(roboto ,20 ,95 , E2 ,16 ,false ,false ,TEXT_ALIGN_LEFT ,black )
   drawText(noto ,20 ,48 , J1 ,14 ,true ,false ,TEXT_ALIGN_LEFT ,black )
   drawText(noto ,20 ,29 , J2 ,14 ,false ,false ,TEXT_ALIGN_LEFT ,black )
end
```

*1 configuration/であってもこのサーチパスの宣言は必要な様です。PC が異なるとパスが見つからないエラーになる事あり。

*2 色の指定例

```
black      = {0, 0, 0, 1}
cyan       = {0, 1, 1, 1}
magenta = {1, 0, 1, 1}
yellow     = {1, 1, 0, 1}
red        = {1, 0, 0, 1}
white      = {1, 1, 1, 1}
```

3.3 音を出す

事前に音源ファイル、例えば sound_shoes0.wav を /configuration/ フォルダに保存しておきます。
lua ファイルの冒頭でそのフォルダにある音源の呼び名を定義します。

＜呼び名＞ = sasl.al.loadSample (＜音源へのルートとファイル＞)

指定の音を一回だけ出す場合
sasl.al.playSample (＜呼び名＞)

isLooping が true の間指定の音を出す場合
sasl.al.playSample (＜呼び名＞ , isLooping)

以下の例では mod_state が 1 になった時に、sound_shoes0 の音を一回だけ出します。

sasl.al.playSample 実行例

```
addSearchResourcesPath ( moduleDirectory .. "/configuration/" )
sound_shoes0 = sasl.al.loadSample ( "sound_shoes0.wav" )
        :
n = 0
function update ( )
        :
   if ( mod_state == 1 ) and ( N == 0 ) then
      sasl.al.playSample ( sound_shoes0 )
      n = 1
   end
        :
end
```

3.4 メニューを追加する

X-Plane 画面上部の Plugins に新規メニューを追加します。メニューを追加できるのは Plugins だけの様です。飛行開始前の設定に使えます。
　　注）以下の記載は main.lua でないとうまく動作しない様です。

設定方法：

(1) メニュー項目を追加する

menuItemID = sasl.appendMenuItem (menuID , string , callback)

- menuItemID : 追加するメニュー項目の呼び名
- menuID : どのメニュー系列に追加するか、そのメニュー系列を指定する。Plugins 系列に追加する場合は PLUGINS_MENU_ID を指定。
- string : 追加するメニューの表示文字列
- callback : そのメニューが選ばれた時に実行する関数、必要な場合のみ記載。

(2) メニュー列を追加する

menuID = sasl.createMenu (string , parentID , parentItemID)

- menuID : 追加するメニュー列の呼び名
- string : 追加するメニュー列の表示文字列、表示しない場合は“”を記入。
- parentID : どの親メニュー列に追加するかを指定する。Plugins 系列に追加する場合は PLUGINS_MENU_ID を指定。
- ParentItemID : 追加する親メニュー項目の呼び名

(3) 特定メニュー項目を選択状態にする

sasl.setMenuItemState (menuID , menuItemID , MenuItemState)

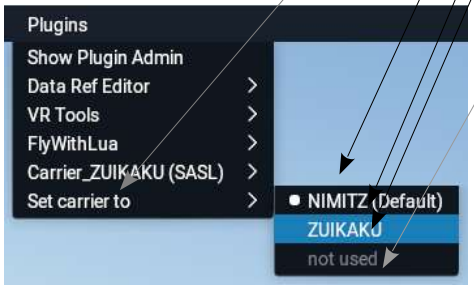
- menuID : 指定するメニュー列の呼び名
- menuItemID : 指定するメニュー項目の呼び名
- state : 指定する状態
  - MENU NO CHECK　初期状態（●印付かない）
  - MENU UNCHECKED　非選択状態（●印付かない））
  - MENU CHECKED　選択状態（●印が付く）

(5) 使えないメニューである事を表示する

sasl.enableMenuItem (menuID , menuItemID , Enable)

- menuID : 指定するメニュー列の呼び名
- menuItemID : 指定するメニュー項目の呼び名
- enabled : 0 : 灰色表示で選択不可能となる
  1 : 通常の表示で選択可能

(以下省略)



以下の例では右図の様に Plugins の下に Set carrier to というサブメニューを追加します。その子供に NIMITZ (Default) , ZUIKAKU, not used の項目を追加し、NIMITZ (Default) は選択状態とします。また not used は選択不可状態とします。

最初に NIMITZ (Default) がクリックされた時実行する関数および ZUIKAKU がクリックされた時実行する関数を記載します。

次に Plugins メニューに Set carrier to というメニューを追加し、これを topMenu とする文を記載します。
作成した topMenu メニューの下に subMenu というメニュー列を追加します。その最初のメニューとして NIMITZ (Default) というメニューを作り、それをクリックしたら changeToNimitz という関数を呼ぶようにします。そしてそのメニューが選択済みである●を表示をします。
2 番目のメニューとして ZUIKAKU というメニューを作り、それをクリックしたら changeToZuikaku という関数を実行する様にします。

設定例：

```
function changeToNimitz ( )
        : (メニュー選択時の処理を記載)
end
function changeToNimitz ( )
        : (メニュー選択時の処理を記載)
end
  :
topMenu = sasl.appendMenuItem (PLUGINS_MENU_ID, "Set carrier to" )
subMenu = sasl.createMenu ("", PLUGINS_MENU_ID, topMenu)
subMenu0= sasl.appendMenuItem(subMenu, "NIMITZ (Default)", changeToNimitz )
sasl.setMenuItemState(subMenu, subMenu0, MENU_CHECKED)
subMenu1= sasl.appendMenuItem(subMenu, "ZUIKAKU" , changeToZuikaku )
subMenu2= sasl.appendMenuItem(subMenu, "not used", changeTo2nd )
sasl.enableMenuItem ( subMenu , subMenu2 , 0)
```

3.5 マウス操作を設定する

SASL でマウス操作を設定する場合は、X-Plane 画面全体の操作設定と contextWindow で指定したエリアのみの操作設定で異なるので注意が必要です。

3.5.1 contextWindow 内のマウス操作設定

最初に main.lua の冒頭で contextWindow の component でマウス操作エリアを position で設定します。指定された component 内 (lua ファイル内）で以下の設定を行います。その設定は contextWindow で指定されたエリア内でのみ機能します。

設定方法：

(1) マウスボタン押下時の設定

```
onMouseDown ( component, x, y, button, parentX, parentY )
      :  ＜マウスボタン押下時の機能を記述＞
   return true
end
```

(2) マウスボタン解放時の設定

```
onMouseUp ( component, x, y, button, parentX, parentY )
      :  ＜マウスボタン解放時の機能を記述＞
   return true
end
```

(3) マウスホイール操作時の設定

```
onMouseWheel ( component, x, y, button, parentX, parentY )
      :  ＜マウスホイール操作時の機能を記述＞
   return true
end
```

(4) マウスボタン保持時の設定

```
onMouseHold ( component, x, y, button, parentX, parentY )
      :  ＜マウスボタン保持時の機能を記述＞
   return true
end
```

(5) マウスドラッグ時の設定

```
onMouseMove ( component, x, y, button, parentX, parentY )
      :  ＜マウスドラッグ時の機能を記述＞
   return true
end
```

（以下省略）

引数の意味

Component	：この関数を記載した component が呼ばれます。
x	：カーソルの X 座標が呼ばれます。
y	：カーソルの Y 座標が呼ばれます。
button	：操作したボタン変数が呼ばれます。 MB_LEFT / MB_RIGHT / MB_MIDDLE
parentX	：カーソルの親の X 座標が呼ばれます。
parentY	：カーソルの親の Y 座標が呼ばれます。

これら関数の最後に return を返し、そのマウス操作を他の component に回さない様にします。

実行例

main.lua 又はその配下の component にて:

```
function onMouseDown (component , x , y , button , parentX , parentY)
   if button == MB_LEFT then print ("MB_LEFT Down")
   elseif button == MB_RIGHT then print ("MB_RIGHT Down")
   end
   return true
end

function onMouseUp (component , x , y , button , parentX , parentY)
   if button == MB_LEFT then print ("MB_LEFT Up")
   elseif button == MB_RIGHT then print ("MB_RIGHT Up")
   end
   return true
end

function onMouseWheel ( component, x, y, button, parentX, parentY, value)
   if value ~= 0 then print ("value= ", value) end
   return true
end
```

3.5.2 X-Plane 画面全体のマウス操作設定

以下の設定は contextWindow で設定されたエリア以外の X-Plane 画面で有効です。設定場所は main.lua または配下の component どちらでも問題ない様です。

以下の機能を使うには component の記述で冒頭に Click System Mode (CSMode) を On にする必要あります。また基本的に下記(2)以降の指定は update 関数やカメラをコントロール関数など連続して状態を監視する関数を使います。

設定方法：

(1) CS モード開始宣言（必須）
＜状態＞ = sasl.setCSMode (mode)　　mode = 0 : Off 1 : On

(2) マウスボタン押下の検出
＜状態＞ = sasl.getCSClickDown (buttonID)　　＜状態＞ = 0 : On 1 : Off

(3) マウスボタン解放の検出
＜状態＞ = sasl.getCSClickUp (buttonID)　　＜状態＞ = 0 : On 1 : Off

(4) マウスボタン押下保持の検出
＜状態＞ = sasl.getCSClickHold (buttonID)　　＜状態＞ = 0 : On 1 : Off

(5) マウスダブルクリックの検出
＜状態＞ = sasl.getCSDoubleClick (buttonID)　　＜状態＞ = 0 : On 1 : Off

(6) マウスホイールのクリック数検出
＜クリック数＞ = sasl.getCSWheelClick (buttonID)

(7) カーソルの X 座標検出（contextWindow 内の座標）
＜X 座標＞ = sasl.getCSMouseXPos ( )

(8) カーソルの Y 座標検出（contextWindow 内の座標）
＜Y 座標＞ = sasl.getCSMouseYPos ( )

(9) カーソルが X-Plane 画面内にあるかを検出
（原則として最初にこの判定を行う事が必要な様です）
＜状態＞ = sasl.getCSCursorOnInterface ( )

（以下省略）

以下の例では contextWindow で interactZone という component を宣言し、その position に指定されたエリアに対するマウス操作を設定します。

main.lua にて：

```
interactZone = contextWindow {
   position        = { 100, 200, 380, 350 } ,
          :
   components  = { InteractZone {
      position     = { 0 , 0 , 380 , 350 } , } ,
   }
}
```

InteractZone.lua にて：

```
sasl.setCSMode(1)
    :
function update()
   if sasl.getCSCursorOnInterface ( ) == 1 then
      if sasl.getCSClickDown ( MB_MIDDLE ) == 1 then
         print ( "MB_MIDDLE Down" )
      end
      if sasl.getCSDragDirection ( ) ~= 0 then
         print ( "Direct", sasl.getCSDragDirection ( ) )
      end
      if sasl.getCSDragValue ( ) ~= 0 then
         print ( "DragValue", sasl.getCSDragValue ( ) )
      end
   end
end
```

3.6 キー操作を設定する

キーボードの各キーに新たな機能を追加出来ます。そのキー操作が影響する範囲がその Component だけなのか、他の component も含むのか、またはプロジェクト全体なのかで設定方法が異なります。
キーボードの既存の機能を使わないでそのプロジェクト専用のキー入力設定を設定する場合は main.lua の冒頭で setInteractivity (false) を宣言します。

(1) 記述した component でのみ有効とする場合

onKeyDown 又は onKeyUP の関数名で下記の引数を使って callback を設定します。キーの押下または解放時にその時の引数を使って callback が実行されます。

設定方法：

① キー押下時の機能を設定

```
onKeyDown ( component, char, Key, shiftDown , ctrlDown, altOptDown)
    :  ＜キー押下時の機能を記述＞
end
```

② キー解放時の機能を設定

```
onKeyUp ( component, char, Key, shiftDown , ctrlDown, altOptDown )
    :  ＜キー解放時の機能を記述＞
end
```

引数の意味

component	: 記述された component そのもの
char	: ASCII コード
key	: virtual key コード
shiftDown	: Shift キー状態　(0:UP, 1:Down)
ctrlDown	: Ctr キー状態　(0:UP, 1:Down)
altOptDown	: Alt キー状態　(0:UP, 1:Down)

(2) 他の component も有効とする場合

この記述はその component のみならず他の component にも有効です。
最初に任意名称(例えば keyHandler)で下記 6 個の引数を持つ関数を使って、キーの機能を定義します。
その後 sasl.registerGlobalKeyHandler ( ) で X-Plane 実行中にそのキーが押下されたらキーの機能を実行する事を定義します。

設定方法：

```
function ＜任意名称＞( char, key, shiftDown , ctrlDown, altOptDown,
event)
    :  ＜callback 機能を記述＞
end
sasl.registerGlobalKeyHandler (＜任意名称＞)
```

引数の意味

char	: ASCII コード	
key	: virtual key コード	
shiftDown	: Shift キー状態　(0:UP, 1:Down)	
ctrlDown	: Ctr キー状態　(0:UP, 1:Down)	
altOptDown	: Alt キー状態　(0:UP, 1:Down)	
event	: キー状態	KB_DOWN_EVENT KB_UP_EVENT KB_HOLD_EVENT

registerGlobalKeyHandler の実行例

```
function keyHandler ( code, key, shiftDown, ctrlDown, altOptDown,
event )
   if (key == SASL_VK_SPACE) and ( event == KB_DOWN_EVENT)
then
    :＜機能を記載＞
   end
end
test = sasl.registerGlobalKeyHandler ( keyHandler )
```

(3) プロジェクト全体を有効とする場合

任意名称の関数で callback を定義し、その関数を registerHotKey(...) で下記の引数を使ってホットキーとして登録します。

設定方法：

```
function ＜任意名称＞ ( )
      :  ＜callback 機能を記述＞
end
＜参照名 1＞ = sasl.registerHotKey (key, shiftDown, ctrlDown, altOptDown,
description, ＜任意名称＞)
```

引数の意味

key	: virtual key コード
shiftDown	: Shift キー状態　(0:UP, 1:Down)
ctrlDown	: Ctrl キー状態　(0:UP, 1:Down)
altOptDown	: Alt キー状態　(0:UP, 1:Down)
description	: 説明文
callback	: callback 関数

他のキーも合わせて操作された時に Callback 処理する場合は次の行も追加する。

```
＜参照名 2＞= sasl.setHotKeyCombination ( ＜参照名 1＞,  key,
shiftDown , ctrlDown , altOptDown )
```

registerHotKey の実行例

```
function anyKey ( )
   print ( " Hello ! " )
end
test1 = sasl.registerHotKey( SASL_VK_T, 1, 1, 0, ”Test”, anyKey )
test2 = sasl.setHotKeyCombination ( test1 , SASL_VK_ENTER , 1 , 1 , 0)
```

表 1　AsciiKeyCode 指定一覧

SASL_KEY_RETURN	SASL_KEY_UP	SASL_KEY_4
SASL_KEY_ESCAPE	SASL_KEY_DOWN	SASL_KEY_5
SASL_KEY_TAB	SASL_KEY_0	SASL_KEY_6
SASL_KEY_DELETE	SASL_KEY_1	SASL_KEY_7
SASL_KEY_LEFT	SASL_KEY_2	SASL_KEY_8
SASL_KEY_RIGHT	SASL_KEY_3	SASL_ KEY_9
		SASL_KEY_DECIMAL

表 2　VirtualKeyCode 指定一覧

SASL_VK_BACK	SASL_VK_A	SASL_VK_NUMPAD5	SASL_VK_F20
SASL_VK_TAB	SASL_VK_B	SASL_VK_NUMPAD6	SASL_VK_F21
SASL_VK_CLEAR	SASL_VK_C	SASL_VK_NUMPAD7	SASL_VK_F22
SASL_VK_RETURN	SASL_VK_D	SASL_VK_NUMPAD8	SASL_VK_F23
SASL_VK_ESCAPE	SASL_VK_E	SASL_VK_NUMPAD9	SASL_VK_F24
SASL_VK_SPACE	SASL_VK_F	SASL_VK_MULTIPLY	SASL_VK_EQUAL
SASL_VK_PRIOR	SASL_VK_G	SASL_VK_ADD	SASL_VK_MINUS
SASL_VK_NEXT	SASL_VK_H	SASL_VK_SEPARATOR	SASL_VK_RBRACE
SASL_VK_END	SASL_VK_I	SASL_VK_SUBTRACT	SASL_VK_LBRACE
SASL_VK_HOME	SASL_VK_J	SASL_VK_DECIMAL	SASL_VK_QUOTE
SASL_VK_LEFT	SASL_VK_K	SASL_VK_DIVIDE	SASL_VK_SEMICOLON
SASL_VK_UP	SASL_VK_L	SASL_VK_F1	SASL_VK_BACKSLASH
SASL_VK_RIGHT	SASL_VK_M	SASL_VK_F2	SASL_VK_COMMA
SASL_VK_DOWN	SASL_VK_N	SASL_VK_F3	SASL_VK_SLASH
SASL_VK_SELECT	SASL_VK_O	SASL_VK_F4	SASL_VK_PERIOD
SASL_VK_PRINT	SASL_VK_P	SASL_VK_F5	SASL_VK_BACKQUOTE
SASL_VK_EXECUTE	SASL_VK_Q	SASL_VK_F6	SASL_VK_ENTER
SASL_VK_SNAPSHOT	SASL_VK_R	SASL_VK_F7	SASL_VK_NUMPAD_ENT
SASL_VK_ INSERT	SASL_VK_S	SASL_VK_F8	SASL_VK_NUMPAD EQ
SASL_VK_DELETE	SASL_VK_T	SASL_VK_F9	
SASL_VK_HELP	SASL_VK_U	SASL_VK_F10	
SASL_VK_0	SASL_VK_V	SASL_VK_F11	
SASL_VK_1	SASL_VK_W	SASL_VK_F12	
SASL_VK_2	SASL_VK_X	SASL_VK_F13	
SASL_VK_3	SASL_VK_Y	SASL_VK_F14	
SASL_VK_4	SASL_VK_Z	SASL_VK_F15	
SASL_VK_5	SASL_VK_NUMPAD0	SASL_VK_F16	
SASL_VK_6	SASL_VK_NUMPAD1	SASL_VK_F17	
SASL_VK_7	SASL_VK_NUMPAD2	SASL_VK_F18	
SASL_VK_8	SASL_VK_NUMPAD3	SASL_VK_F19	
SASL_VK_9	SASL_VK_NUMPAD4		

3.7 カメラ設定

設定方法：

(1) カメラの位置、向き、倍率を指定します。

sasl.setCamera (x , y , z , pitch , yaw , roll , zoom)

引数の意味

x	: OpenGL の x 座標
y	: OpenGL の y 座標
z	: OpenGL の z 座標
pitch	: 上下向き (+:上向き, -:下向き)
yaw	: 左右向き (+:右向き, -:左向き)
roll	: 光軸回りの回転
zoom	: 画面の拡大縮小 (0.0 : 最小, 1.0 : 最大)

(2) 設定用関数で配置されたカメラの呼び名称を宣言する。

＜カメラ名称＞ = sasl.registerCameraController (＜設定用関数＞)

＜設定用関数＞: callback 関数

(3) 指定された呼び名称のカメラを開始する

sasl.startCameraControl (＜カメラ名称＞ , Status)

＜カメラ ID＞	: virtual key コード
Status	: 以下のいずれかを指定する。
	CAMERA_NOT_CONTROLLED
	CAMERA_CONTROLLED_UNTIL_VIEW_CHANGE
	CAMERA_CONTROLLED_ALWAYS

(4) 現在可動しているカメラを停止する

sasl.stopCameraControl ( )

3.7.1 マウス制御カメラの設定

機体近くの点を中心にカメラをマウスドラッグで回転させマウスホイールで画面を拡大縮小した例を以下にします。

```
local planeX = globalPropertyf("sim/flightmodel/position/local_x")
local planeY = globalPropertyf("sim/flightmodel/position/local_y")
local planeZ = globalPropertyf("sim/flightmodel/position/local_z")
local planePsi = globalPropertyf("sim/flightmodel/position/psi")
local planeThe = globalPropertyf("sim/flightmodel/position/theta")
DragDir = 0
DragVal = 0
Psi = 0.0
The = 0.0
PsiLast = 0.0
TheLast = 0.0
zoomC = 1.0
sasl.setCSMode(1)

function CameraController0 ( )
   if sasl.getCSCursorOnInterface()==1 and sasl.getCSClickHold(MB_LEFT)==1 then
      DragDir = sasl.getCSDragDirection ( ) * 0.017444
      DragVal = sasl.getCSDragValue ( )
      Psi = 0.5 * DragVal * math.cos ( DragDir )
      The = 0.5 * DragVal * math.sin ( DragDir )
   end

   if sasl.getCSCursorOnInterface ( ) ==1 and sasl.getCSClickUp (MB_LEFT) ==1 then
      PsiLast = Psi
      TheLast = The
   end

   p = get(planePsi) + PsiLast + Psi
   q = TheLast + The
   if q > 24 then q = 24 end
   if q < -55 then q = -55 end
   prad = p * 0.017444
   qrad = q * 0.017444
   x = get(planeX) - 6 * math.sin(prad)
   y = get(planeY) - 6 * math.sin(qrad) + 2.5
   z = get(planeZ) + 6 * math.cos(prad)
   sasl.setCamera(x , y , z , q , p , 0.0 , zoomC )
end
controllerID0 = sasl.registerCameraController(CameraController0)
sasl.startCameraControl(controllerID0, CAMERA_CONTROLLED_ALWAYS)
      :
```

```
function update ( )
   if sasl.getCSCursorOnInterface ( ) == 1 and sasl.getCSWheelClicks ( ) ~=0 then
      zoomC = zoomC + 0.2*sasl.getCSWheelClicks ( )
      if zoomC < 1.0 then zoomC = 1.0 end
   end
end
```

3.7.2 カメラ切り替え

マウス中央ボタン押下でカメラを切り替える例を以下に示します。

事前に controllerID0、controllerID1 のカメラ呼び名でカメラを登録してあるとします。

実行例

```
currentCamera == 0

function flipCamera ( )
   currentCamera = 1 – currentCamera
      if currentCamera == 0 then
         sasl.stopCameraControl ( controllerID0 )
         sasl.startCameraControl ( controllerID1 , CAMERA_CONTROLLED_ALWAYS )
      elseif currentCamera == 1 then
         sasl.stopCameraControl ( controllerID1 )
         sasl.startCameraControl ( controllerID0 , CAMERA_CONTROLLED_ALWAYS )
      end
end


function update ( )
  if sasl.getCSCursorOnInterface ( ) == 1 and sasl.getCSClickDown ( MB_MIDDLE ) == 1 then
      flipCamera()
   end
end
```

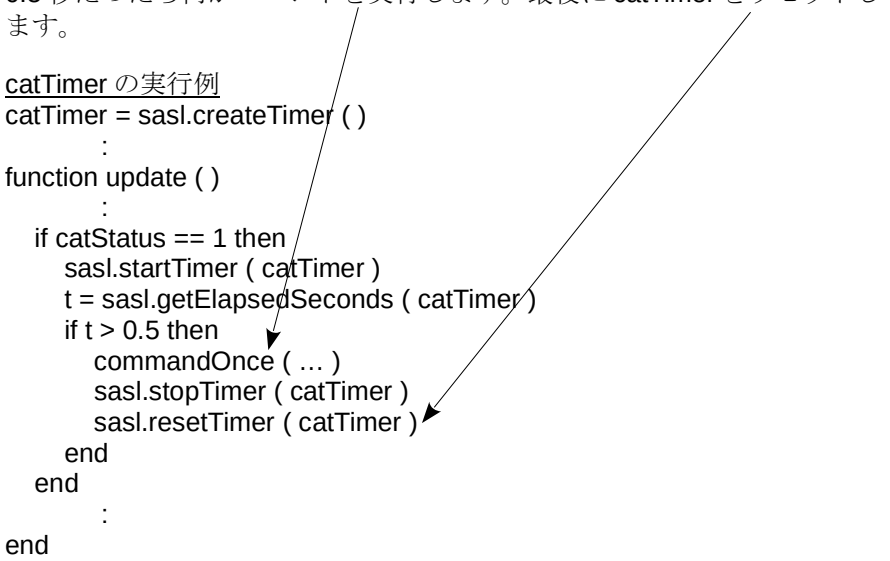
3.8 タイマーを使う

Lua 実行中にある操作を特定の時間遅らせたい場合があります。その場合はタイマーを使います。
最初に createTimer 関数 (引数なし) でタイマー名を定義します。

設定方法：

<タイマー名> = sasl.createTimer ( )

その後 update 関数の中でタイマーを起動時に sasl.startTimer (<タイマー名>) を実行し、sasl.getElapsedSeconds (<タイマー名>) で経過時間を取得し、所定の時間が来たら必要な操作を行います。
以下の例では、catStatus が 1 になった時 catTimer のカウントを開始し、0.5 秒たったら何かコマンドを実行します。最後に catTimer をリセットします。



catTimer の実行例

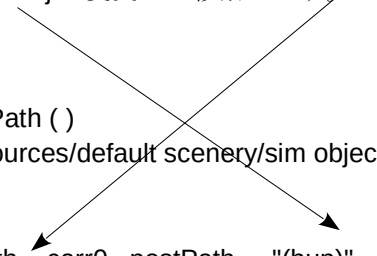
```
catTimer = sasl.createTimer ( )
          :
function update ( )
          :
   if catStatus == 1 then
      sasl.startTimer ( catTimer )
      t = sasl.getElapsedSeconds ( catTimer )
      if t > 0.5 then
         commandOnce ( … )
         sasl.stopTimer ( catTimer )
         sasl.resetTimer ( catTimer )
      end
   end
          :
end
```

3.9 ファイル移動

Lua の OS 機能を使ってファイルの移動やファイル名称変更する事が出来ます。
ファイル指定にはファイルパスも含まれるので、ファイルパスが異なれば
ファイルを移動する事になります。その記述は次の通りです。

設定方法：
print (os.rename (<元のファイル>, <変更後のファイル>))

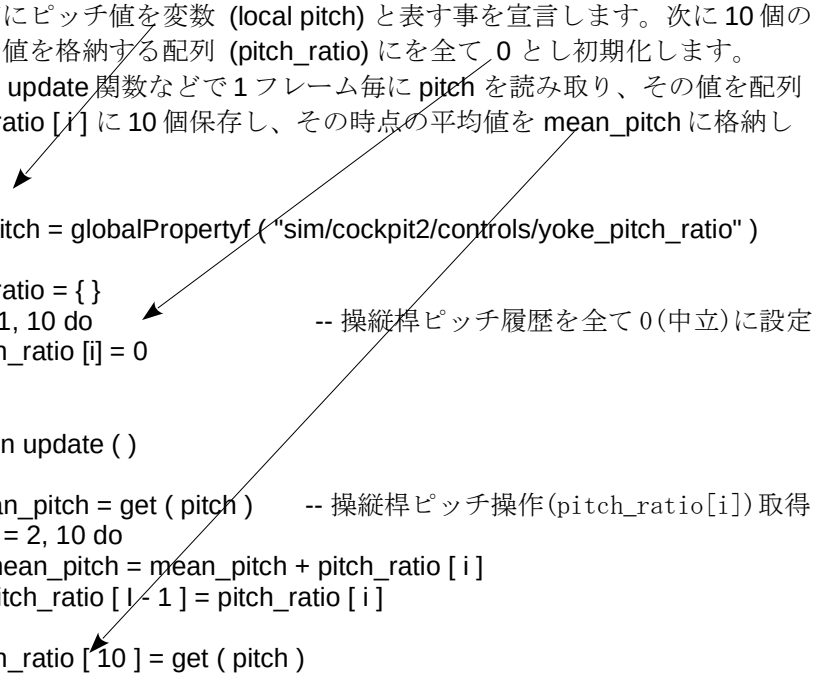
以下の例では、prePath で指定したフォルダにある Nimitz.obj を postPath に
ファイル名を (bup)Nimitz.obj に変更して移動します。



ファイル移動例
```
carr0 = "Nimitz.obj"
xppath = sasl.getXPlanePath ( )
prePath = xppath .. "Resources/default scenery/sim objects/dynamic/"
postPath = <目的のパス>

print ( os.rename ( prePath .. carr0 , postPath  .. "(bup)" .. carr0 ) )
```

3.10 ジョイスティック出力平均化

ジョイスティクの操作軸出力は雑音が大きいのでその値を平均化する必要
があります。
以下にジョイスティックのピッチ値を平均化する例を上げます。Lua 記述
の最初にピッチ値を変数 (local pitch) と表す事を宣言します。次に 10 個の
ピッチ値を格納する配列 (pitch_ratio) にを全て 0 とし初期化します。
その後 update 関数などで 1 フレーム毎に pitch を読み取り、その値を配列
pitch_ratio [i] に 10 個保存し、その時点の平均値を mean_pitch に格納し
ます。



```
local pitch = globalPropertyf ( "sim/cockpit2/controls/yoke_pitch_ratio" )

pitch_ratio = { }
for i = 1, 10 do                          -- 操縦桿ピッチ履歴を全て 0(中立)に設定
   pitch_ratio [i] = 0
end

function update ( )
     :
   mean_pitch = get ( pitch )       -- 操縦桿ピッチ操作(pitch_ratio[i])取得
   for i = 2, 10 do
      mean_pitch = mean_pitch + pitch_ratio [ i ]
      pitch_ratio [ i - 1 ] = pitch_ratio [ i ]
   end
   pitch_ratio [ 10 ] = get ( pitch )
   mean_pitch = mean_pitch / 10                  -- 操縦桿ピッチの 10 回平均
(mean_pitch)
     :
end
```

3.11 コマンド実行

対象となるコマンドを findCommand 関数で宣言し変数を設定します。次にそのコマンドを実行する関数 (この例では changeToZuikaku) を記述します。下記の例では preCar が postCarr でない時に pushAcf で定義されたコマンドを実行します。

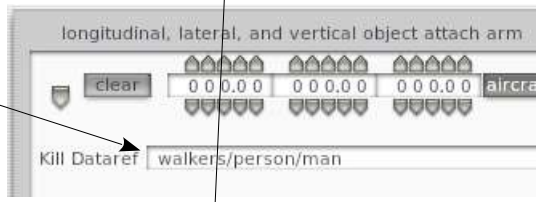
```
pushAcf = findCommand ( "sim/operation/Carrier_Catshot" )

function changeToZuikaku ( )
    :
  if preCarr ~= postCarr then
     commandOnce ( pushAcf )
  end
    :
end
```

3.12 killdataref の使用

最初に当該 Plugin 専用の dataref を決めます。他の Plugin と名称が重なってはいけません。sim/...の名称は使えません。/を含む名称とする事の様です。

dataref を createGlobalProperty 関数を使って宣言します。{2.4 項(5)参照}
その後、その dataref でオブジェクトを表示させたい場面で set 関数で dataref の値を 0 に設定します。その設定は X-Plane 内に記憶され、飛行を停止しても残ります。

PlaneMaker の Standard/Misc Objects 画面で表示させたいオブジェクトの killdataref 欄にその dataref を記載します。



下記の例ではメニュー操作などで changeToMan ( ) が実行された場合 reload_scenery が実行されるので、有効なオブジェクト(manIsSet) が使われ飛行を開始します。

記載例

```
local manIsSet = createGlobalPropertyi ( "walkers/person/man", 0 )
local womanIsSet = createGlobalPropertyi ( "walkers/person/woman", 1 )
reloadScn =  sasl.findCommand ( "sim/operation/reload_scenery" )
    :
function changeToMan()
   set( manIsSet , 0 )                    -- killdataref of man is active
   set( womanIsSet , 1 )
   sasl.commandOnce ( reloadScn )
end
```

4. デバック

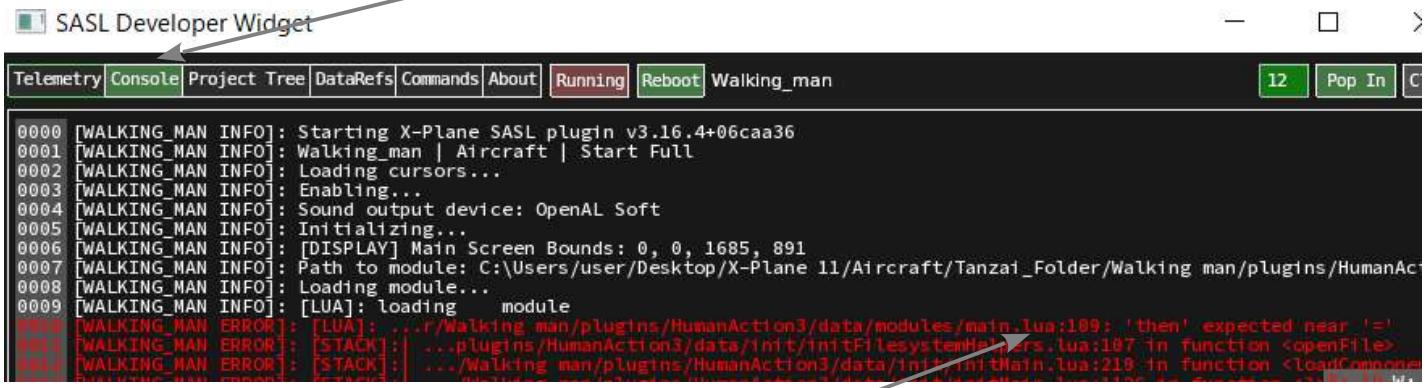
4.1 Widgit の見かた

Widgit を表示しデバックに使う方法を以下に示します。

(1) 2.2 項の configuration.ini ファイルで Widgit: 欄に 1 を設定します。

(2) 対象機体を立ち上げ、画面上部の Plugins メニューリストから <機体名>(SASL)タブを選択し、Show Widgit をクリックします。

(3) 下図の画面が出るのでその中から **Console** タブを選択します。



(4) 同画面のリスト欄に赤色の表示が有ればエラーが発生しています。
・エラーのあった lua ファイル名およびその行番号が表示される場合があります。
・lua ファイルの変数を確認したい場合は、その lua 記述の途中に例えば

```
print ( "cat_state = ", cat_state )
```

を記述するとその値がリスト欄に表示されるので参考にします。

・または下記の複数行コメントを非実行にする方法を使って動作確認する場合も有ります。

```
--[[
複数行のコメント
]]
```